Data concurrency

Presented by DB2 Developer Domain

http://www7b.software.ibm.com/dmdd/

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

What this tutorial is about

This tutorial will introduce you to the concept of data consistency and to the various mechanisms that are used by DB2 Universal Database to maintain database consistency in both single- and multi-user environments. In this tutorial, you will learn:

- What data consistency is
- What transactions are and how they are initiated and terminated
- How transactions are isolated from each other in a multi-user environment
- How DB2 Universal Database provides concurrency control through the use of locks
- What types of locks are available and how locks are acquired
- What factors influence locking

This tutorial is the sixth in a series of six tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Fundamentals Certification (Exam 700). The material in this tutorial primarily covers the objectives in Section 6 of the exam, entitled "Data Concurrency." You can view these objectives at: *http://www.ibm.com/certify/tests/obj700.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Edition.

Terminology review

In order to understand some of the material presented in this tutorial, you should be familiar with the following terms:

- **Object:** Anything in a database that can be created or manipulated with SQL (e.g., tables, views, indexes, packages).

- **Table:** A logical structure that is used to present data as a collection of unordered rows with a fixed number of columns. Each column contains a set of values, each value of the same data type (or a subtype of the column's data type); the definitions of the columns make up the table structure, and the rows contain the actual table data.

- **Record:** The storage representation of a row in a table.

- **Field:** The storage representation of a column in a table.

- **Value:** A specific data item that can be found at each intersection of a row and

column in a database table.

- **Structured Query Language (SQL):** A standardized language used to define objects and manipulate data in a relational database. (For more on SQL, see the *fourth tutorial in this series*.

- **Call-Level Interface (CLI):** A callable Application Programming Interface (API) that is used as an alternative to SQL. In contrast to embedded SQL, CLI does not require precompiling or binding by the user, but instead provides a standard set of functions that process SQL statements and perform related services at application run time.

- **DB2 optimizer:** A component of the SQL precompiler that chooses an access plan for a Data Manipulation Language (DML) SQL statement by modeling the execution cost of several alternative access plans and choosing the one with the minimal estimated cost.

About the author

Roger E. Sanders is a database performance engineer with Network Appliance, Inc. You can contact him at *rsanders@netapp.com*. Roger has written several computer magazine articles, presented at three International DB2 User's Group (IDUG) conferences, is the co-author of an IBM Redbook, and is the author of *All-In-One DB2 Administration Exam Guide, DB2 Universal Database SQL Developer's Guide, DB2 Universal Database API Developer's Guide, DB2 Universal Database CLI Developer's Guide, ODBC 3.5 Developer's Guide,* and the *Developer's Handbook to DB2 for Common Servers.* He also holds the following professional certifications:

- IBM Certified Advanced Database Administrator - DB2 Universal Database V8.1 for Linux, UNIX, and Windows
- IBM Certified Database Administrator- DB2 Universal Database V8.1 for Linux, UNIX, and Windows
- IBM Certified Developer - DB2 Universal Database V8.1 Family
- IBM Certified Database Associate - DB2 Universal Database V8.1 Family

Section 2. Transactions

Understanding data consistency

What is data consistency? The best way to answer this question is by looking at an example. Suppose your company owns a chain of restaurants and you have a database that is designed to keep track of supplies stored at each of those restaurants. To facilitate the supply-purchasing process, your database contains an inventory table for each restaurant in the chain. Whenever supplies are received or used by an individual restaurant, the corresponding inventory table for that restaurant is modified to reflect the changes.

Now, suppose some bottles of ketchup are physically moved from one restaurant to another. In order to accurately represent this inventory move, the ketchup bottle count value stored in the donating restaurant's table needs to be lowered and the ketchup bottle count value stored in the receiving restaurant's table needs to be raised. If a user lowers the ketchup bottle count in the donating restaurant's inventory table but fails to raise the ketchup bottle count in the receiving restaurant's inventory table, the data will become *inconsistent.* Now the total ketchup bottle count for the chain of restaurants is no longer accurate.

Data in a database can become inconsistent if a user forgets to make all necessary changes (as in our restaurant example), if the system crashes while the user is in the middle of making changes, or if a database application for some reason stops prematurely. Inconsistency can also occur when several users are accessing the same database tables at the same time. In an effort to prevent data inconsistency, particularly in a multi-user environment, the developers of DB2 Universal Database incorporated the following data consistency support mechanisms into its design:

- Transactions
- Isolation levels
- Locks

We'll discuss each in turn in the following panels.

Transactions and transaction boundaries

A *transaction* (otherwise known as a *unit of work*) is a recoverable sequence of one or more SQL operations grouped together as a single unit, usually within an application process. The initiation and termination of a transaction defines the points of database consistency; either the effects of all SQL operations performed within a transaction are applied to the database, or the effects of all SQL operations performed are completely undone and thrown away.

With embedded SQL applications and scripts that are run from the Command Center, the Script Center, or the Command Line Processor, transactions are automatically

initiated the first time an executable SQL statement is executed, either after a connection to a database been established or after an existing transaction has been terminated. Once initiated, a transaction must be explicitly terminated by the user or application that initiated it, unless a process known as *automatic commit* is being used (in which case each individual SQL statement submitted for execution is treated as a single transaction that is implicitly committed as soon as it is executed).

In most cases, transactions are terminated by executing either the `COMMIT` or the `ROLLBACK` statement. When you execute the `COMMIT` statement, all changes that you have made to the database since the transaction was initiated are made permanent -- that is, they are *committed.* When you execute the `ROLLBACK` statement, all changes that you've made to the database since the transaction was initiated are backed out and the database is returned -- or *rolled back* -- to the state it was in before the transaction began. In either case, the database is guaranteed to be returned to a consistent state at the completion of the transaction.

It is important to note that, while transactions provide generic database consistency by ensuring that changes to data only become permanent after a transaction has been successfully committed, it is up to the user or application to ensure that the sequence of SQL operations within each transaction will always result in a consistent database.

Effects of COMMIT and ROLLBACK operations

As we noted in the last panel, transactions are usually terminated by executing either the `COMMIT` or the `ROLLBACK` SQL statement. To understand how each of these statements work, it helps to look at an example.

Imagine that we execute the following SQL statements in the order shown:

```
CONNECT TO MY_DB
CREATE TABLE DEPARTMENT (DEPT_ID INTEGER NOT NULL, DEPT_NAME VARCHAR(20))
INSERT INTO DEPARTMENT VALUES(100, 'PAYROLL')
INSERT INTO DEPARTMENT VALUES(200, 'ACCOUNTING')
COMMIT

INSERT INTO DEPARTMENT VALUES(300, 'SALES')
ROLLBACK

INSERT INTO DEPARTMENT VALUES(500, 'MARKETING')
COMMIT
```

A table named DEPARTMENT will be created, and it will look something like this:

DEPT_ID	**DEPT_NAME**
100	PAYROLL
200	ACCOUNTING
500	MARKETING

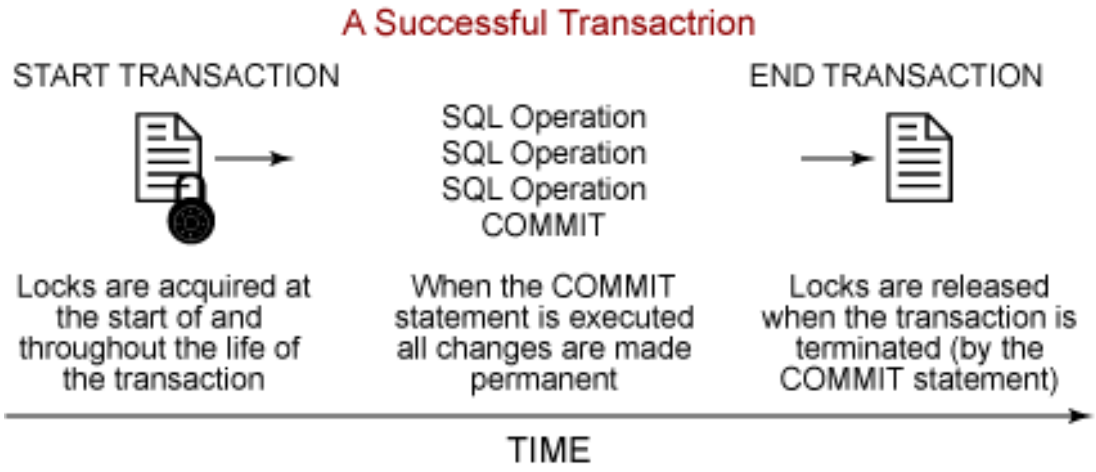
That's because when we execute the first `COMMIT` statement, the creation of the table named DEPARTMENT, along with the insertion of two records into the DEPARTMENT table, will be made permanent. On the other hand, when we execute the `ROLLBACK` statement, the third record inserted into the DEPARTMENT table is removed and the table is returned to the state it was in before the insert operation was performed. Finally, when we execute the second `COMMIT` statement, the insertion of the fourth record into the DEPARTMENT is made permanent and the database is again returned to a consistent state.

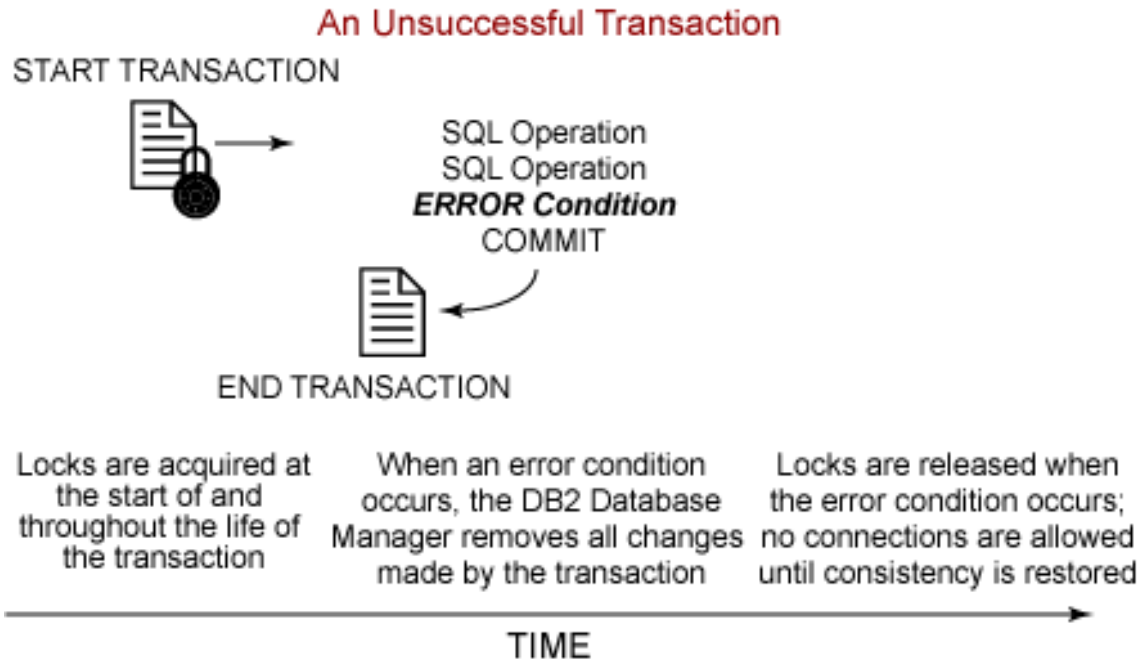
As you can see from this example, a commit or rollback operation only affects changes that are made within the transaction that the commit or rollback operation ends. As long as data changes remain uncommitted, other users and applications are usually unable to see them (there are exceptions, which we will look at later), and they can be backed out with a rollback operation. Once data changes are committed, however, they become accessible to other users and applications and can no longer be removed by a rollback operation.

Effects of an unsuccessful transaction

We have just seen what happens when a transaction is terminated by a `COMMIT` or a `ROLLBACK` statement. But what happens if a system failure occurs before a transaction can be completed? In these situations the DB2 Database Manager will back out all uncommitted changes in order to restore the database consistency that it assumes existed when the transaction was initiated. This is done through the use of transaction log files, which contain information about each SQL statement executed by a transaction, along with information about whether or not that transaction was successfully committed or rolled back.

The following illustration compares the effects of a successful transaction with those of a transaction that fails before it can be successfully terminated:


A Successful Transactrion
START TRANSACTION
SQL Operation
SQL Operation
SQL Operation
COMMIT
END TRANSACTION
Locks are acquired at the start of and throughout the life of the transaction
When the COMMIT statement is executed all changes are made permanent
Locks are released when the transaction is terminated (by the COMMIT statement)
TIME


An Unsuccessful Transaction
START TRANSACTION
SQL Operation
SQL Operation
ERROR Condition
COMMIT
END TRANSACTION
Locks are acquired at the start of and throughout the life of the transaction
When an error condition occurs, the DB2 Database Manager removes all changes made by the transaction
Locks are released when the error condition occurs; no connections are allowed until consistency is restored
TIME

Section 3. Concurrency and isolation levels

Phenomena that can occur when multiple users access a database

With single-user databases, each transaction is executed serially and does not have to contend with interference from other transactions. In a multi-user database environment, however, transactions may execute simultaneously, and each has the potential to interfere with any other transaction that is running. When transactions are not isolated from each other in multi-user environments, four types or phenomena can occur:

- **Lost update:** This event occurs when two transactions read and attempt to update the same data, and one of the updates is lost. For example: Transaction 1 and Transaction 2 read the same row of data and both calculate new values for that row based upon the data read. If Transaction 1 updates the row with its new value and Transaction 2 updates the same row, the update operation performed by Transaction 1 is lost. Because of the way it has been designed, DB2 Universal Database does not allow this type of phenomenon to occur.

- **Dirty read:** This event occurs when a transaction reads data that has not yet been committed. For example: Transaction 1 changes a row of data and Transaction 2 reads the changed row before Transaction 1 has committed the change. If Transaction 1 rolls back the change, Transaction 2 will have read data that is not considered to have ever existed.

- **Nonrepeatable read:** This event occurs when a transaction reads the same row of data twice, but gets different data values each time. For example: Transaction 1 reads a row of data and Transaction 2 changes or deletes that row and commits the change. When Transaction 1 attempts to reread the row, it will retrieve different data values (if the row was updated) or discover that the row no longer exists (if the row was deleted).

- **Phantom:** This event occurs when a row of data that matches search criteria is not seen initially, but then seen in a later read operation. For example: Transaction 1 reads a set of rows that satisfy some search criteria and Transaction 2 inserts a new row that matches Transaction 1's search criteria. If Transaction 1 re-executes the query that produced the original set of rows, a different set of rows will be retrieved.

Maintaining database consistency and data integrity, while allowing more than one application to access the same data at the same time, is known as *concurrency.* One of the ways DB2 Universal Database attempts to enforce concurrency is through the use of *isolation levels,* which determine how data used in one transaction is locked or isolated from other transactions while the first transaction accesses it. DB2 Universal Database uses the following isolation levels to enforce concurrency:

- Repeatable Read

- Read Stability
- Cursor Stability
- Uncommitted Read

We'll discuss each in turn in the following panels.

The Repeatable Read isolation level

When the *Repeatable Read* isolation level is used, all rows referenced by a single transaction are locked for the duration of that transaction. With this isolation level, any SELECT statement that is issued more than once within the same transaction will always yield the same results; lost updates, dirty reads, nonrepeatable reads, and phantoms cannot occur.

Transactions using the Repeatable Read isolation level can retrieve the same set of rows multiple times and perform any number of operations on them until terminated by a commit or a rollback operation; other transactions are not allowed to perform insert, update, or delete operations that will affect the set of rows being used as long the isolating transaction exists. To guarantee that the data being accessed by a transaction running under the Repeatable Read isolation level is not adversely affected by other transactions, each row referenced by the isolating transaction is locked -- not just the rows that are actually retrieved and/or modified. Thus, if a transaction scans 1,000 rows but only retrieves 10, locks are acquired and held on all 1,000 rows scanned, not just on the 10 rows retrieved.

So how does this isolation level work in a real-world situation? Suppose you own a large hotel and you have a Web site that allows individuals to reserve rooms on a first-come, first-served basis. If your hotel reservation application runs under the Repeatable Read isolation level, whenever a customer retrieves a list of all rooms available for a given range of dates, you will not be able to change the room rate for those rooms during the date range specified, and other customers will not be able to make or cancel reservations that would cause the list to change, until the transaction that generated the list is terminated. (You can change room rates, and other customers can make or cancel room reservations, for any room or date that falls outside the range specified by the first customer's query.)

The Read Stability isolation level

When the *Read Stability* isolation level is used, all rows that are retrieved by a single transaction are locked for the duration of that transaction. When this isolation level is used, each row read by the isolating transaction cannot be changed by other transactions until the isolating transaction terminates. In addition, changes made to other rows by other transactions will not be seen by a transaction running under the Read Stability isolation level until they have been committed. Therefore, when the Read Stability isolation level is used, SELECT statements that are issued more than

once within the same transaction may not always yield the same results. Lost updates, dirty reads, and nonrepeatable reads cannot occur; phantoms, on the other hand, can and may be seen.

With the Repeatable Read isolation level, each row that is referenced by the isolating transaction is locked; however, under the Read Stability isolation level, only the rows that the isolating transaction actually retrieves and/or modifies are locked. Thus, if a transaction scans 1,000 rows but only retrieves 10, locks are only acquired and held on the 10 rows retrieved -- not on all 1,000 rows scanned.

So how does this isolation level change the way our hotel reservation application works? Now, when a customer retrieves a list of all rooms available for a given range of dates, you will be able to change the room rate for any room in the hotel, and other customers will be able to cancel room reservations for rooms that had been reserved for the date range specified by the first customer's query. Therefore, if the list is generated again before the transaction that submitted the query is terminated, the new list produced may contain new room rates or rooms that were not available when the list was first produced.

The Cursor Stability isolation level

When the *Cursor Stability* isolation level is used, each row that is referenced by a cursor being used by the isolating transaction is locked as long as the cursor is positioned on that row. The lock acquired remains in effect either until the cursor is repositioned (usually by calling the `FETCH` statement) or until the isolating transaction terminates. Thus, when this isolation level is used, `SELECT` statements that are issued more than once within the same transaction may not always yield the same results. Lost updates and dirty reads cannot occur; nonrepeatable reads and phantoms, however, can and may be seen.

When a transaction using the Cursor Stability isolation level retrieves a row from a table via an updatable cursor, no other transaction can update or delete that row while the cursor is positioned on it. However, other transactions can add new rows to the table and perform update and/or delete operations on rows positioned on either side of the locked row, provided that the locked row itself was not accessed using an index. Furthermore, if the isolating transaction modifies any row it retrieves, no other transaction can update or delete that row until the isolating transaction is terminated, even after the cursor is no longer positioned on the modified row.

Transactions using the Cursor Stability isolation level will not see changes made to other rows by other transactions until those changes have been committed. By default, the Cursor Stability isolation level is the isolation level used by most transactions.

How does this isolation level affect our hotel reservation application? Now, when a customer retrieves a list of all rooms available for a given range of dates and then views information about each room on the list produced (one room at a time), you will be able to change the room rate for any room in the hotel, and other customers will be able to make or cancel reservations for any room, over any date range; the only exception is the room the first customer is currently examining. When the first customer

views information about another room in the list, the same is true for this new room; you will now be able to change the room rate and other customers will be able to make reservations for the previous room that the first customer was examining, but not for the current one.

The Uncommitted Read isolation level

When the *Uncommitted Read* isolation level is used, rows that are retrieved by a single transaction are only locked for the duration of that transaction if another transaction attempts to drop or alter the table from which the rows were retrieved. Because rows often remain unlocked when this isolation level is used, lost updates, dirty reads, nonrepeatable reads, and phantoms can occur.

In most cases, transactions using the Uncommitted Read isolation level can see changes made to rows by other transactions before those changes are committed or rolled back. However, such transactions can neither see nor access tables, views, or indexes that have been created by other transactions until those transactions have been committed. Likewise, transactions using the Uncommitted Read isolation level will only learn that an existing table, view, or index has been dropped by another transaction when that transaction terminates. There is one exception to this behavior: when a transaction running under the Uncommitted Read isolation level uses an updatable cursor, that transaction will behave as if it is running under the Cursor Stability isolation level and the constraints of the Cursor Stability isolation level will be applied.

The Uncommitted Read isolation level is commonly used for transactions that access read-only tables and/or transactions that execute `SELECT` statements on which uncommitted data from other transactions will have no adverse affect.

So how does this isolation level work with our hotel reservation application? Now, when a customer retrieves a list of all rooms available for a given range of dates, you will be able to change the room rate for any room in the hotel, and other customers will be able to make or cancel reservations for any room, over any date range. Furthermore, the list produced can contain rooms that other customers have chosen to cancel reservations for, even if they've not yet terminated their transaction and committed those cancellations to the database.

Specifying the isolation level

Although isolation levels control how resources are locked for transactions, they are actually specified at the application level. For embedded SQL applications, the isolation level to be used is specified at precompile time, or when the application is bound to a database. In most cases, the isolation level for applications written in a supported compiled language (such as C or C++) is set via the `ISOLATION` option of the `PRECOMPILE PROGRAM` and `BIND` commands/APIs. For Call Level Interface (CLI) applications, the isolation level to be used is set at application run time by calling the

SQLSetConnectAttr() function with the SQL_ATTR_TXN_ISOLATION connection attribute specified. Isolation levels for CLI applications can also be set by assigning a value to the TXNISOLATION keyword, which can be found in the db2cli.ini configuration file. With JDBC and SQLJ applications, the isolation level is set at application run time by calling the setTransactionIsolation() method that resides within the java.sql connection interface.

When no isolation level is specified, the Cursor Stability isolation level is used by default. This is true for commands and scripts that are executed from the Command Line Processor (CLP) as well as for embedded SQL, CLI, JDBC, and SQLJ applications. Thus, the isolation level used by commands and scripts run from the CLP can also be specified; in this case, the isolation level to be used is set by executing the CHANGE ISOLATION command within the CLP before making a connection to a database.

Choosing the proper isolation level

Choosing the appropriate isolation level to use for a transaction is very important. The isolation level not only influences how well the database supports concurrency; it also affects the overall performance of the application containing the transaction. That's because the resources required to acquire and free locks vary with each isolation level.

Generally, when more restrictive isolation levels are used, less concurrency support is provided and overall performance may be slowed because more resources are acquired and held. However, when you are deciding on the best isolation level to use, you should make your decision by determining which phenomena are acceptable and which are not. The following heuristic can be used to help you decide which isolation level to use for a particular situation:

- If you are executing queries on read-only databases, or if you are executing queries and do not care if uncommitted data values are returned, use the Uncommitted Read isolation level. (Read-only transactions needed -- high data stability not required.)

- If you want maximum concurrency without seeing uncommitted data values, use the Cursor Stability isolation level. (Read/write transactions needed -- high data stability not required.)

- If you want concurrency and you want qualified rows to remain stable for the duration of an individual transaction, use the Read Stability isolation level. (Read-only or read/write transactions needed -- high data stability required.)

- If you are executing queries and do not want to see changes made to the result data sets produced, use the Repeatable Read isolation level. (Read-only transactions needed -- extremely high data stability required.)

Section 4. Locks

How locking works

In the section on Concurrency and isolation levels on page 8 , we saw that DB2 Universal Database isolates transactions from each other through the use of *locks.* A lock is a mechanism that is used to associate a data resource with a single transaction, with the purpose of controlling how other transactions interact with that resource while it is associated with the owning transaction. (The transaction that a locked resource is associated with is said to *hold* or *own* the lock.) The DB2 Database Manager uses locks to prohibit transactions from accessing uncommitted data written by other transactions (unless the Uncommitted Read isolation level is used) and to prohibit the updating of rows by other transactions when the owning transaction is using a restrictive isolation level. Once a lock is acquired, it is held until the owning transaction is terminated; at that point, the lock is released and the data resource is made available to other transactions.

If one transaction attempts to access a data resource in a way that is incompatible with the lock being held by another transaction (we'll look at lock compatibility shortly), that transaction must wait until the owning transaction has ended. This is known as a *lock wait.* When a lock wait event occurs, the transaction attempting to access the data resource simply stops execution until the owning transaction has terminated and the incompatible lock is released.

Lock attributes

All locks have the following basic attributes:

- **Object:** The *object* attribute identifies the data resource that is being locked. The DB2 Database Manager acquires locks on data resources, such as tablespaces, tables, and rows, whenever they are needed.

- **Size:** The *size* attribute specifies the physical size of the portion of the data resource that is being locked. A lock does not always have to control an entire data resource. For example, rather than giving an application exclusive control over an entire table, the DB2 Database Manager can give an application exclusive control over a specific row in a table.

- **Duration:** The *duration* attribute specifies the length of time for which a lock is held. A transaction's isolation level usually controls the duration of a lock.

- **Mode:** The *mode* attribute specifies the type of access allowed for the lock owner as well as the type of access permitted for concurrent users of the locked data resource. This attribute is commonly referred to as the *lock state.*

Lock states: Types of locks

The state of a lock determines the type of access allowed for the lock owner as well as the type of access permitted for concurrent users of a locked data resource. The following list identifies the lock states that are available, in order of increasing control:

Lock state (Mode):	Intent None (IN)
Applicable objects:	Tablespaces and tables
Description:	The lock owner can read data in the locked table, including uncommitted data, but cannot change this data. In this mode, the lock owner does not acquire row-level locks; therefore, other concurrent applications can read and change data in the table.
Lock state (Mode):	Intent Share (IS)
Applicable objects:	Tablespaces and tables
Description:	The lock owner can read data in the locked table, but cannot change this data. Again, because the lock owner does not acquire row-level locks, other concurrent applications can both read and change data in the table. (When a transaction owns an Intent Share lock on a table, it acquires a Share lock on each row it reads.) This lock is acquired when a transaction does not convey the intent to update rows in the table.
Lock state (Mode):	Next Key Share (NS)
Applicable objects:	Rows
Description:	The lock owner and all concurrent transactions can read, but cannot change, data in the locked row. This lock is acquired in place of a Share lock on data that is read using the Read Stability or Cursor Stability transaction isolation level.
Lock state (Mode):	Share (S)
Applicable objects:	Tables and rows
Description:	The lock owner and any other concurrent transactions can read, but cannot change, data in the locked table or row. As long as a table is not Share locked, individual rows in that table can be Share locked. If, however, a table is Share locked, row-level Share locks in that table cannot be acquired by the lock owner. If either a table or a row is Share locked, other concurrent transactions can read the data, but they cannot change it.
Lock state (Mode):	Intent Exclusive (IX)
Applicable objects:	Tablespaces and tables
Description:	The lock owner and any other concurrent applications can read and

change data in the locked table. When the lock owner reads data from the table, it acquires a Share lock on each row it reads, and it acquires both an Update and an Exclusive lock on each row it updates. Other concurrent applications can both read and update the locked table. This lock is acquired when a transaction conveys the intent to update rows in the table. (The `SELECT FOR UPDATE`, `UPDATE ... WHERE`, and `INSERT` statements convey the intent to update.)

Lock state (Mode): Share With Intent Exclusive (SIX)

Applicable objects: Tables

Description: The lock owner can both read and change data in the locked table. The lock owner acquires Exclusive locks on the rows it updates but does not acquire locks on rows that it reads; therefore, other concurrent applications can read but cannot update the data in the locked table.

Lock state (Mode): Update (U)

Applicable objects: Tables and rows

Description: The lock owner can update data in the locked table and the lock owner automatically acquires Exclusive locks on any rows it updates. Other concurrent applications can read but cannot update the data in the locked table.

Lock state (Mode): Next Key Exclusive (NX)

Applicable objects: Rows

Description: The lock owner can read but cannot change the locked row. This lock is acquired on the next row in a table when a row is deleted from or inserted into the index for that table.

Lock state (Mode): Next Key Weak Exclusive (NW)

Applicable objects: Rows

Description: The lock owner can read but cannot change the locked row. This lock is acquired on the next row in a table when a row is inserted into the index of a noncatalog table.

Lock state (Mode): Exclusive (X)

Applicable objects: Tables and rows

Description: The lock owner can both read and change data in the locked table or row. If an Exclusive lock is acquired, only applications using the Uncommitted Read isolation level are allowed to access the locked table or row(s). Exclusive locks are acquired for data resources that are going to be manipulated with the `INSERT`, `UPDATE`, and/or `DELETE` statements.

Lock state (Mode): Weak Exclusive (WE)

Applicable objects:	Rows
Description:	The lock owner can read and change the locked row. This lock is acquired on a row when it is inserted into a noncatalog table.

Lock state (Mode):	Super Exclusive (Z)
Applicable objects:	Tablespaces and tables
Description:	The lock owner can alter a table, drop a table, create an index, or drop an index. This lock is automatically acquired on a table whenever a transaction attempts to perform any one of these operations. No other concurrent transactions are allowed to read or update the table until this lock is removed.

Lock compatibility

If the state of one lock placed on a data resource enables another lock to be placed on the same resource, the two locks (or states) are said to be *compatible.* Whenever one transaction holds a lock on a data resource and a second transaction requests a lock on the same resource, the DB2 Database Manager examines the two lock states to determine whether or not they are compatible. If the locks are compatible, the lock is granted to the second transaction (provided no other transaction is waiting for the data resource). If however, the locks are incompatible, the second transaction must wait until the first transaction releases its lock before it can gain access to the resource and continue processing. (If there is more than one incompatible lock in place, the second transaction must wait until all of them are released.) Refer to the *IBM DB2 Universal Database Administration Guide: Performance* documentation (or search the DB2 Information Center for *Lock type compatibility* topics) for specific information on which locks are compatible with one another and which are not.

Lock conversion

When a transaction attempts to access a data resource that it already holds a lock on, and the mode of access needed requires a more restrictive lock than the one already held, the state of the lock held is changed to the more restrictive state. The operation of changing the state of a lock already held to a more restrictive state is known as *lock conversion.* Lock conversion occurs because a transaction can hold only one lock on a data resource at a time.

In most cases, lock conversion is performed for row-level locks and the conversion process is pretty straightforward. For example, if a Share (S) or an Update (U) row-level lock is held and an Exclusive (X) lock is needed, the held lock will be converted to an Exclusive (X) lock. Intent Exclusive (IX) locks and Share (S) locks are special cases, however, since neither is considered to be more restrictive than the other. Thus, if one of these row-level locks is held and the other is requested, the held

lock is converted to a Share with Intent Exclusive (SIX) lock. Similar conversions result in the requested lock state becoming the new lock state of the held lock, provided the requested lock state is more restrictive. (Lock conversion only occurs if a held lock can increase its restriction.) Once a lock's state has been converted, the lock stays at the highest state obtained until the transaction holding the lock is terminated.

Lock escalation

All locks require space for storage; because the space available is not infinite, the DB2 Database Manager must limit the amount of space that can be used for locks (this is done through the `maxlocks` database configuration parameter). In order to prevent a specific database agent from exceeding the lock space limitations established, a process known as *lock escalation* is performed automatically whenever too many locks (of any type) have been acquired. Lock escalation is the conversion of several individual row-level locks within the same table to a single table-level lock. Since lock escalation is handled internally, the only externally detectable result might be a reduction in concurrent access on one or more tables.

Here's how lock escalation works: When a transaction requests a lock and the lock storage space is full, one of the tables associated with the transaction is selected, a table-level lock is acquired on its behalf, all row-level locks for that table are released (to create space in the lock list data structure), and the table-level lock is added to the lock list. If this process does not free up enough space, another table is selected and the process is repeated until enough free space is available. At that point, the requested lock is acquired and the transaction resumes execution. However, if the necessary lock space is still unavailable after all the transaction's row-level locks have been escalated, the transaction is asked (via an SQL error code) to either commit or rollback all changes that have been made since its initiation and the transaction is terminated.

Deadlocks

Contention for locks by two or more transactions can sometimes result in a situation known as a *deadlock.* The best way to illustrate how a deadlock can occur is by example: Suppose Transaction 1 acquires an Exclusive (X) lock on Table A and Transaction 2 acquires an Exclusive (X) lock on Table B. Now, suppose Transaction 1 attempts to acquire an Exclusive (X) lock on Table B and Transaction 2 attempts to acquire an Exclusive (X) lock on Table A. Processing by both transactions will be suspended until their second lock request is granted. However, because neither lock request can be granted until one of the transactions releases the lock it currently holds (by performing a commit or rollback operation), and because neither transaction can release the lock it currently holds (because both are suspended and waiting on locks), the transactions are stuck in a deadlock situation.

When a deadlock situation occurs, all transactions involved will wait indefinitely for a lock to be released, unless some outside agent takes action. DB2 Universal

Database's tool for handling deadlocks is an asynchronous system background process, known as the *deadlock detector.* The sole responsibility of the deadlock detector is to locate and resolve any deadlocks found in the locking subsystem. The deadlock detector stays asleep most of the time, but wakes up at preset intervals to determine whether or not a deadlock situation exists. If the deadlock detector discovers a deadlock in the locking subsystem, it selects, terminates, and rolls back one of the transactions involved. (The transaction that is terminated and rolled back receives an SQL error code and all locks it had acquired are released.) Usually, the remaining transaction(s) can then proceed.

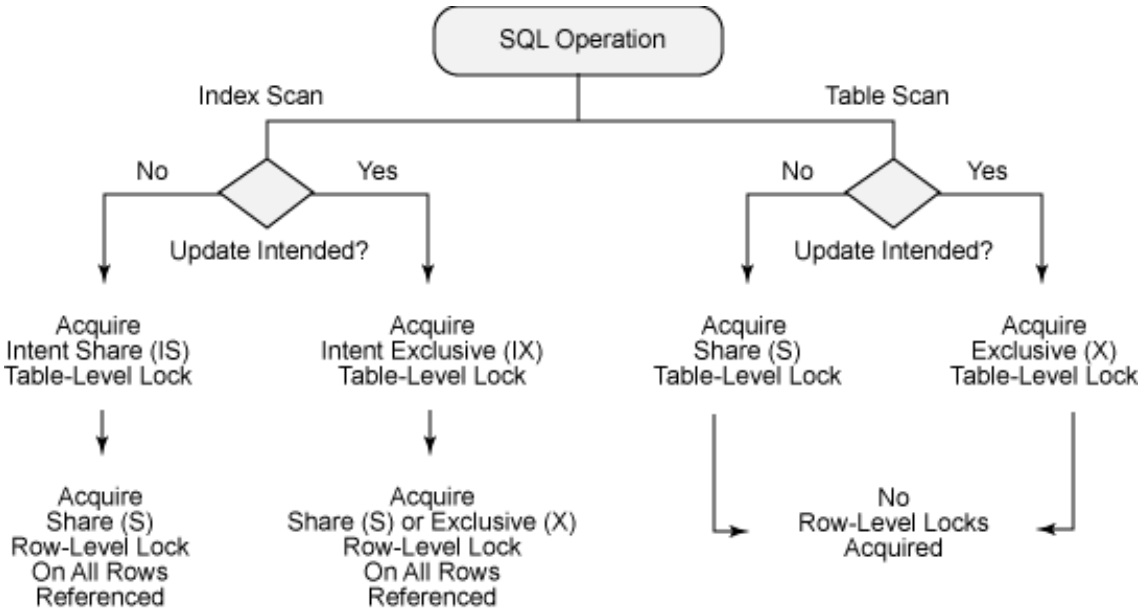
Lock timeouts

Any time a transaction holds a lock on a particular data resource (a table or row, for example), other transactions may be denied access to that resource until the owning transaction terminates and frees all the locks it has acquired. Without some sort of lock timeout detection mechanism in place, a transaction might wait indefinitely for a lock to be released. Such a situation might occur, for example, when a transaction is waiting for a lock that is held by another user's application to be released, and the other user has left his or her workstation without performing some interaction that would allow the application to terminate the owning transaction. Obviously, these types of situations can cause poor application performance. To avoid stalling other applications when these types of situations occur, a lock timeout value can be specified in a database's configuration file (via the `locktimeout` database configuration parameter). When used, this value controls the amount of time any transaction will wait to obtain a requested lock. If the desired lock is not acquired before the time interval specified elapses, the waiting application receives an error and the transaction requesting the lock is rolled back. Distributed transaction application environments are particularly prone to these sorts of timeouts; you can avoid them by using lock timeouts.

How locks are acquired

In most cases, the DB2 Database Manager implicitly acquires locks as they are needed, and these locks remain under the DB2 Database Manager's control. Except in situations where the Uncommitted Read isolation level is used, a transaction never needs to explicitly request a lock. In fact, the only database object that can be explicitly locked by a transaction is a table object.

The following illustration shows the logic that is used to determine which type of lock to acquire for a referenced object:



The DB2 Database Manager always attempts to acquire row-level locks. However, this behavior can be modified by executing a special form of the `ALTER TABLE` statement, as follows:

```
ALTER TABLE [TableName] LOCKSIZE TABLE
```

where *`TableName`* identifies the name of an existing table for which all transactions are to acquire table-level locks for when accessing it.

The DB2 Database Manager can also be forced to acquire a table-level lock on a table for a specific transaction by executing the `LOCK TABLE` statement, as follows:

```
LOCK TABLE [TableName] IN [SHARE | EXCLUSIVE] MODE
```

where *`TableName`* identifies the name of an existing table for which a table-level lock is to be acquired (provided that no other transaction has an incompatible lock on this table). If this statement is executed with the `SHARE` mode specified, a table-level lock that will allow other transactions to read, but not change, the data stored in it will be acquired; if executed with the `EXCLUSIVE` mode specified, a table-level lock that does not allow other transactions to read or modify data stored in the table will be acquired.

Concurrency and granularity

As we mentioned earlier, any time a transaction holds a lock on a particular data resource, other transactions may be denied access to that resource until the owning transaction terminates. Therefore, to optimize for maximum concurrency, row-level locks are usually better than table-level locks, because they limit access to a much smaller resource. However, because each lock acquired requires some amount of storage space and processing time to manage, a single table-level lock will require less overhead than several individual row-level locks. Unless otherwise specified, row-level

locks are acquired by default.

The *granularity* of locks (that is, whether row-level locks or table-level locks are acquired) can be controlled through the use of the `ALTER TABLE ... LOCKSIZE TABLE`, `ALTER TABLE ... LOCKSIZE ROW`, and `LOCK TABLE` statements. The `ALTER TABLE ... LOCKSIZE TABLE` statement provides a global approach to granularity that results in table-level locks being acquired by all transactions that access rows within a particular table. On the other hand, the `LOCK TABLE` statement allows table-level locks to be acquired at an individual transaction level. When either of these statements are used, a single Share (S) or Exclusive (X) table-level lock is acquired whenever a lock is needed. As a result, overall performance is usually improved, since only one table-level lock must be acquired and released instead of several different row-level locks.

Section 5. Factors that influence locking

Transaction processing

From a locking standpoint, all transactions typically fall under one of the following categories:

- **Read-Only:** This refers to transactions that contain `SELECT` statements (which are intrinsically read-only), `SELECT` statements that have the `FOR READ ONLY` clause specified, or SQL statements that are ambiguous, but are presumed to be read-only because of the `BLOCKING` option specified as part of the precompile and/or bind process.

- **Intent-To-Change:** This refers to transactions that contain `SELECT` statements that have the `FOR UPDATE` clause specified, or SQL statements that are ambiguous, but are presumed to be intended for change because of the way they are interpreted by the SQL precompiler.

- **Change:** This refers to transactions that contain `INSERT`, `UPDATE`, and/or `DELETE` statements, but not `UPDATE ... WHERE CURRENT OF ...` or `DELETE ... WHERE CURRENT OF ...` statements.

- **Cursor-Controlled:** This refers to transactions that contain `UPDATE ... WHERE CURRENT OF ...` and `DELETE ... WHERE CURRENT OF ...` statements.

Read-Only transactions typically use Intent Share (IS) and/or Share (S) locks. Intent-To-Change transactions, on the other hand, use Update (U), Intent Exclusive (IX), and Exclusive (X) locks for tables, and Share (S), Update (U), and Exclusive (X) locks for rows. Change transactions tend to use Intent Exclusive (IX) and/or Exclusive (X) locks, while Cursor Controlled transactions often use Intent Exclusive (IX) and/or Exclusive (X) locks.

Data access paths

When an SQL statement is precompiled, the DB2 optimizer explores various ways to satisfy that statement's request and estimates the execution cost involved for each approach. Based on this evaluation, the DB2 optimizer then selects what it believes to be the optimal access plan. (The access plan specifies the operations required and the order in which those operations are to be performed to resolve an SQL request.) An access plan can use one of two ways to access data in a table: by directly reading the table (which is known as performing a *table* or a *relation scan*), or by reading an index on that table and then retrieving the row in the table to which a particular index entry refers (which is known as performing an *index scan*).

The access path chosen by the DB2 optimizer, which is often determined by the database's design, can have a significant impact on the number of locks acquired and the lock states used. For example, when an index scan is used to locate a specific row, the DB2 Database Manager will most likely acquire one or more Intent Share (IS) row-level locks. However, if a table scan is used, because the entire table must be scanned, in sequence, to locate a specific row, the DB2 Database Manager may opt to acquire a single Share (S) table-level lock.

Section 6. Summary

Summary

This tutorial was designed to introduce you to the concept of data consistency and to the various mechanisms that are used by DB2 Universal Database to maintain database consistency in both single- and multi-user environments. A database can become inconsistent if a user forgets to make all necessary changes, if the system crashes while a user is in the middle of making changes, or if a database application for some reason stops prematurely. Inconsistency can also occur when several users are accessing the same database tables at the same time. For example, one user might read another user's changes before all tables have been properly updated and take some inappropriate action or make an incorrect change based on the premature data values read. In an effort to prevent data inconsistency, particularly in a multi-user environment, the developers of DB2 Universal Database incorporated the following data consistency support mechanisms into its design:

- Transactions
- Isolation levels
- Locks

A transaction (otherwise known as a unit of work) is a recoverable sequence of one or more SQL operations that are grouped together as a single unit, usually within an application process. The initiation and termination of a transaction define the points of database consistency; either the effects of all SQL operations performed within a transaction are applied to the database, or the effects of all SQL operations performed are completely undone and thrown away. In either case, the database is guaranteed to be in a consistent state at the completion of each transaction.

Maintaining database consistency and data integrity, while allowing more than one application to access the same data at the same time, is known as concurrency. With DB2 Universal Database, concurrency is enforced through the use of isolation levels. Four different isolation levels are available:

- Repeatable Read
- Read Stability
- Cursor Stability
- Uncommitted Read

Along with isolation levels, DB2 Universal Database provides concurrency in multi-user environments through the use of locks. A lock is a mechanism that is used to associate a data resource with a single transaction, with the purpose of controlling how other transactions interact with that resource while it is associated with the transaction that owns the lock. Twelve different types of lock are available:

- Intent None (IN)
- Intent Share (IS)
- Next Key Share (NS)

- Share (S)
- Intent Exclusive (IX)
- Share with Intent Exclusive (SIX)
- Update (U)
- Next Key Exclusive (NX)
- Next Key Weak Exclusive (NW)
- Exclusive (X)
- Weak Exclusive (W)
- Super Exclusive (Z)

To maintain data integrity, the DB2 Database Manager acquires locks implicitly, and all locks acquired remain under the DB2 Database Manager's control. Locks can be placed on tablespaces, tables, and rows.

To optimize for maximum concurrency, row-level locks are usually better than table-level locks, because they limit access to a much smaller resource. However, because each lock acquired requires some amount of storage space and processing time to manage, a single table-level lock will require less overhead than several individual row-level locks.

Resources

For more information on DB2 Universal Database and database concurrency:

- *IBM DB2 Universal Database Administration Guide: Performance, Version 8*, SC09-4821-00. International Business Machines Corporation, 1993-2002.
- *All-In-One DB2 Administration Exam Guide* . Sanders, Roger E., McGraw-Hill/Osborne, 2002.

For more information on the DB2 Family Fundamentals Exam 700:

- *IBM Data Management Skills* information.
- Download a *self-study course for experienced database administrators (DBAs)* to quickly and easily gain skills in DB2 UDB.
- Download a *self study course for experienced relational database programmers* who would like to know more about DB2.
- *General Certification Information*, including some book suggestions, exam objectives, courses

Check out the other parts of the DB2 V8.1 Family Fundamentals Certification Prep series:

- *DB2 V8.1 Family Fundamentals Certification Prep, Part 1 of 6: DB2 Planning*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 2 of 6: DB2 Security*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 3 of 6: Accessing DB2 UDB*

Data

- *DB2 V8.1 Family Fundamentals Certification Prep, Part 4 of 6: Working with DB2 UDB Data*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 5 of 6: Working with DB2 UDB Objects*

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT style sheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.